Équation de Schrödinger

1 Résolution de l'équation de Schrödinger indépendante du temps par la méthode dite « Shooting method » .

1.1 Quelques fonctions utiles

1. Pour écrire la fonction `rechercheIndice`, comme la complexité demandée est simplement linéaire, on se contentera d'un parcours avec une boucle `while`. L'hypothèse de l'énoncé fait que l'on est sûr de trouver un tel indice.
   Une proposition de code possible est la suivante :

```
def rechercheIndice(u, u0):
  i = 0
  while u[i] <= u0: # Au pire u[len(u)-1] > u0...
      i = i + 1
  return i - 1
```

   Dans le pire des cas on parcourt la totalité du tableau, et dans chaque itération on fait un nombre fini d'opérations élémentaires en temps constant (comparaison, consultation d'un élément du tableau, affectation, addition). Dès lors la complexité est bien linéaire.

2. Comme le tableau est trié on peut avantageusement écrire une méthode de recherche dichotomique. La complexité devient alors logarithmique.

3. L'équation cartésienne de la droite passant par (x_0, y_0) et (x_1, y_1) est $Y = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(X - x_0) = \frac{(y_1 - y_0)X + y_0x_1 - y_1x_0}{x_1 - x_0}$
   Dès lors on peut proposer le code suivant :

```
def interpolationLineaire(x0, y0, x1, y1, x):
  return ((y1 - y0) * x + y0 * x1 - y1 * x0) / (x1 - x0)
```

4. L'écriture de la fonction **valeurApprochee** ne pose pas de problème. Dans un premier temps on recherche l'indice `i` du plus grand élément du tableau `u` inférieur ou égal à `u0`, puis on utilise la fonction d'interpolation linéaire entre les points qui entourent `u0`, à savoir `(u[i],val[i])` et `(u[i+1],val[i+1])`.
   On peut donc proposer le code suivant :

```
def valeurApprochee(u, valf, u0):
   i = indice(u, u0)
  return interpolationLinaire(u[i],valf[i],u[i+1],valf[i+1],u0)
```

1.2 Probabilité de présence

1. Le principe d'intégration numérique de la méthode dite des rectangles à gauche d'une fonction g sur un intervalle $[a, b]$ repose sur un découpage de l'intervalle en N sous-intervalles de longueur $\frac{b-a}{N}$ et une approximation de l'aire située sous la courbe par l'aire d'un rectangle de hauteur égale à la valeur de la fonction g prise en la borne inférieure de chaque sous-intervalle.

On a alors $\int_a^b g(x)x$ approchée par $\frac{b-a}{N}\sum_{k=0}^{N-1} g\left(a+k\frac{b-a}{N}\right)$.

2. L'adaptation de cette méthode consiste surtout à tenir compte du fait que la subdivision de l'intervalle n'est pas nécessairement régulière.

   On peut alors proposer le script suivant qui calcule dans A une approximation de l'intégrale recherchée

```
A = 0
n = len(u)
for i in range(n-1):
    A = A + (u[i+1]-u[i])*valf[i]**2
```

1.3 Résolution d'une équation différentielle

1. La principale différence vient d'une modification du pas h au cours de l'algorithme. On fait évoluer ce pas en comparant une évaluation de l'erreur commise à un seuil reçu en paramètre (il y a donc un paramètre en plus).

   Ceci fait qu'il y a certaines itérations lors desquelles on n'enregistre pas de données.

2. Le calcul $t = t+h$ semble intervenir trop tard. En effet dans une itération donnée, quand on passe dans le cas (e) c'est que l'on a jugé que le pas h était satisfaisant. Dès lors il faudrait enregistrer les données pour l'instant t correspondant. Or ici, après le calcul de y on modifie h puis t avant l'enregistrement des données. Il faudrait donc placer ce calcul entre celui de y et de h.

3. Ici quand on rentre dans la méthode on ne sait pas combien de valeurs vont être calculées car le pas évolue. Dès lors il vaut mieux prendre une structure dont on pourra faire évoluer le nombre d'éléments, ce qui sera le cas pour une liste, et pas pour un tableau.

   C'est surtout en termes de complexité que cela pose problème. L'ajout d'un élément en dernière position à une liste se fait en temps constant. L'augmentation de la taille d'un tableau d'un élément est linéaire en la taille du tableau (ce qui se traduit par une complexité au minimum quadratique uniquement pour cette opération...) Rien n'empêche pour utilisation ultérieure (et pour gagner un peu de temps) de convertir les listes en tableau ensuite..

4. Sachant que la valeur de ϵ estime l'erreur commise dans ce schéma numérique (et qu'elle est quadratique en le pas), expliquer le principe de la méthode d'**Euler-Richadson** et l'idée sous-jacente à l'évolution du pas.

5. On peut remarquer que le calcul $h = \frac{0{,}9\times h}{\sqrt{\alpha}}$ est toujours fait, que l'on soit dans le cas (d) ou le cas (e). Par ailleurs le cas $\alpha = 1$ n'est pas traîté dans la description de l'algorithme. Même s'il est très peu probable qu'après des calculs sur les flottants on tombe sur ce cas, il vaut mieux prévoir quelque chose. On peut l'inclure dans la cas où on diminue le pas (car comme ça on ne prend pas de risque...) Dès lors on peut proposer le code suivant :

```
from math import sqrt

def EulerRichardson(y0, h, T, seuil):
  liste_y = [y0]
  liste_t = [0]
  tc = 0
  yc = y0
  pas = h
  while tc <= T:
      k = F(tc, yc)
      kp = F(tc + pas / 2, yc + k * pas / 2)
      epsilon = pas / 2 * abs(kp - k)
      alpha = epsilon / seuil

      if alpha < 1:
```

```
            yc = yc + pas * k
            tc = tc + pas
            liste_y.append(yc)
            liste_t.append(tc)

        pas = pas * 0.9 / sqrt(alpha)
    return [liste_t, liste_y]
```

6. Expliquer clairement, sans réécrire la totalité du code, comment adapter votre fonction pour résoudre une équation différentielle du second ordre (l'équation de Schrödinger réduite par exemple). On peut se ramener à une équation du premier ordre en introduisant un vecteur $Y = (y, y')$. Il faudra bien sûr que la fonction F soit adaptée pour prendre en entrée le temps t et le vecteur Y et rende en sortie Y'. Il faudra également passer deux conditions initiales à la fonction de résolution de l'équation différentielle.

1.4 Recherche dichotomique d'une solution

1. Avec nos techniques numériques de calcul l'intégrale aura toujours une valeur finie (éventuellement grande, voire trop grande pour être représentée par un flotant en machine). On n'aura ainsi jamais la certitude de la divergence de l'intégrale... Par ailleurs il semble difficile d'utiliser une méthode d'intégration numérique sur un intervalle infini, car cela prendrait nécessairement un temps infini...
2. Pour une solution paire on a nécessairement $f'(0) = 0$. Par ailleurs si on trouve une solution normalisable pour $f(0) = 1$, on pourra la normer en la multipliant par la bonne constante. Donc une recherche avec ce couple de conditions initiales est suffisant. Pour chercher les solutions impaires on prendra en revanche $f(0) = 0$ et $f'(0) = 1$.
3. Le schéma de recherche est le suivant : Résoudre l'équation de Schrödinger pour $e = e_{min}$ qui rend la solution $f_{e_{min}}$. Calculer $A = f_{e_min} = 0,9 \times u_{max}$. Résoudre l'équation de Schrödinger pour $e = e_{max}$ qui rend la solution $f_{e_{max}}$. Calculer $B = f_{e_{max}} = 0,9 \times u_{max}$. Résoudre l'équation de Schrödinger pour $e = \frac{e_{min}+e_{max}}{2}$ qui rend la solution $f_{candidate}$. Calculer $M = f_{candidate} = 0,9 \times u_max$. Tant que $|M| > seuil_{div}$ faire Si $A * M > 0$: $e_{min} = e, A = M$ sinon $e_{max} = e, B = M$ Résoudre l'équation de Schrödinger pour $e = \frac{e_{min}+e_{max}}{2}$ qui rend la solution $f_{candidate}$. Calculer $M = f_{candidate} = 0,9 \times u_max$.

4.
```
f rechercheDichotomique(emin, emax, umax, seuil_div):
On suppose h et seuil ces valeurs connues
Résolution avec les deux énergies emin et emax etles conditions initiales pour une
   solution paire
      (u, y, yp) = EulerRichardson(1, 0, h, umax, seuil,emin)
      A = approche(u, y, 0.9 * umax)
      (u, y, yp) = EulerRichardson(1, 0, h, umax, seuil,emax)
      B = approche(u, y, 0.9 * umax)
      inf = emin
      sup = emax
      e = (inf + sup) / 2
      (u, y, yp) = EulerRichardson(1, 0, h, umax, seuil, e)
      M = approche(u, y, 0.9 * umax)
      while abs(M) > seuil_div:
              if A * M < 0:
                      sup = e
                      B = M
              else:
                      inf = e
                      A = M
              e = (inf + sup) / 2
              (u, y, yp) = EulerRichardson(1, 0, h, umax, seuil,e)
```

```
            M = approche(u, y, 0.9 * umax)
    return [e, u, y]
```

Remarque : il n'est en fait pas nécessaire avec les hypothèses de l'énoncé de refaire le calcul de la solution pour emax, ni de calculer B ni même de mettre à jour A et B dans la boucle while !

5. On ne peut bien sûr pas appeler cette fonction avec un seuil de divergence $seuil_{div}$ aussi petit que l'on veut afin d'obtenir autant de chiffres significatifs que l'on veut, car il existe un plus petit flottant représentable en machine, ce qui limite la précision numérique des calculs menés.
6. Il s'agit d'une « shooting method » car en gros on essaie de résoudre avec une énergie quelconque (comme si on lançait un caillou vers une cible) et on ajuste cette énergie jusqu'à ce qu'il n'y ait plus divergence (jusqu'à ce que l'on touche la cible).

1.5 Détermination du nombre de nœuds d'une solution

1. ```
f nbPassagesParZero(valf):
    n = len(valf)
    indice = 0 # Indice du dernier franchissement
    nb = 0 # Nombre de passages par zéro
    if valf[0] == 0: # Cas où le premier élément du tableau est nul. On ne le
   compte pas comme un franchissement
            signe_courant_positif = valf[1] > 0
    else:
            signe_courant_positif = valf[0] > 0
    for i in range(1, n):
            if (signe_courant_positif and valf[i] <= 0) or (not
   signe_courant_positif and valf[i] >= 0) :
                    nb = nb + 1
                    signe_courant_positif = not signe_courant_positif
                    indice = i
turn nb, indic
```

2. ```
f estMonotoneApresDernierZero (valf, indice):
    n = len(valf)
    i = indice
    monotone = True
    # On détermine le sens de variation à l'aide de valf
    croissant = valf[indice] > valf[indice - 1]
            while i < n - 1 and monotone:
                    monotone = (croissant and valf[i + 1] >= valf[i]) or (not
   croissant and valf[i + 1] <= valf[i])
                    i = i + 1
turn monotone
```

3. Pour les solutions paires, si on note $nbNoeuds$ son nombre de noeuds, et $nbAnnulations$ le nombre d'annulation de la solution approchée sur $\mathscr{R}^{+\star}$, on a $nbNoeuds = 2 \times nbAnnulations$ si la solution approchée n'est pas monotone après sa dernière annulation, et $nbNoeuds = 2 \times (nbAnnulations - 1)$ dans le cas contraire. Pour les solutions impaires on a, avec les mêmes notations, $nbNoeuds = 1 + 2 \times nbAnnulations$ si la solution approchée n'est pas monotone après sa dernière annulation, et $nbNoeuds = 1 + 2 \times (nbAnnulations - 1)$ dans le cas contraire.

4. ```
f nbNoeuds(valf):
    impaire = valf[0] == 0
    [nb, indice] = nbPassagesParZero(valf)
    if estMonotoneApresDernierZero (valf, indice):
            nb = nb - 1 # Le dernier franchissement n'est pas un vrai noeud
```
```

```
    noeuds = 2 * nb # Pour l'aspect symétrique
    if impaire:
            noeuds = noeuds + 1 # On rajoute le noeud en 0
    return noeuds
```

5. Quelle démarche proposeriez-vous pour déterminer toutes les solutions de l'équation de Schrödinger ayant une énergie comprise entre deux bornes e_{min} et e_{max} ? On ne codera pas la fonction correspondante.